Perspective on Parallel Programming

Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

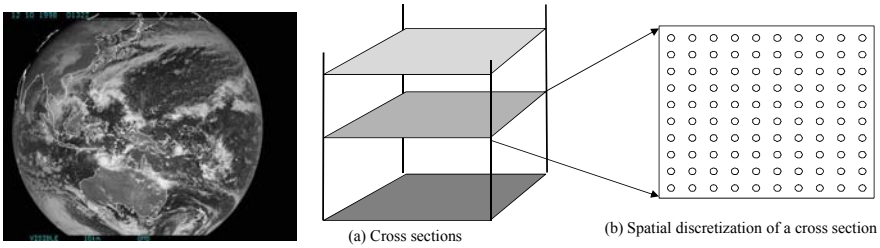
Outline of this Presentation

- Motivating Problems
  - Wide range of applications
  - Scientific, engineering, and commercial computing problems
  - Ocean Currents, Galaxy Evolution, Ray Tracing, and Data Mining
- The parallelization process
  - Decomposition, Assignment, Orchestration, and Mapping
- Simple example on the Parallelization Process
  - Orchestration under three major parallel programming models
  - What primitives must a system support?

Why Bother with Parallel Programs?

- They are what runs on the parallel machines
  - Make better design decisions and system tradeoffs
- Led to the key advances in uniprocessor architecture
  - Caches and instruction set design
- More important in multiprocessors
  - New degrees of freedom, greater penalties for mismatch
- Algorithm designers
  - Design parallel algorithms that will run well on real systems
- Programmers
  - Understand optimization issues and obtain best performance
- Architects
  - Understand workloads which are valuable for design and evaluation

Simulating Ocean Currents

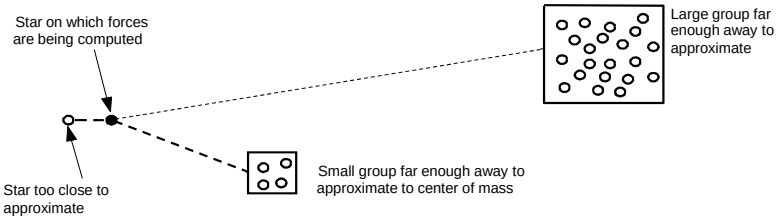


- Simulates the motion of water currents in the ocean
  - Influence of atmospheric effects, wind, and friction with ocean floor and walls
- Model as two-dimensional cross section grids
  - Discretization in space (grid points) and time (finite time-steps)
  - Finer spatial and temporal resolution => greater accuracy
  - Equations of motion are setup and solved at all grid points in one time step
  - Concurrency across and within grid computations

Simulating the Evolution of Galaxies

- Simulate the interactions of many stars evolving over time
- Computing forces on each star under the effect of all other stars
  - Expensive $O(n^2)$ brute force approach
  - Hierarchical Methods take advantage of force law
    - **Barnes-Hut algorithm O (n log n)**

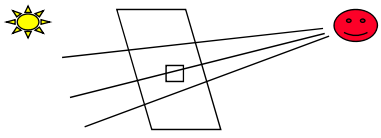
$$\frac{G\, m_1\, m_2}{r^2}$$



- Many time-steps, plenty of concurrency across stars

Rendering Scenes by Ray Tracing

- Scene is represented as a set of objects in 3D space
- Image being rendered is represented as 2D array of pixels
- Shoot rays from a specific viewpoint through image and into scene
  - Compute ray reflection, refraction, and lighting interactions
  - They bounce around as they strike objects
  - They generate new rays => ray tree per input ray



- Result is color, opacity, and brightness for each pixel
- Parallelism across rays

Creating a Parallel Program

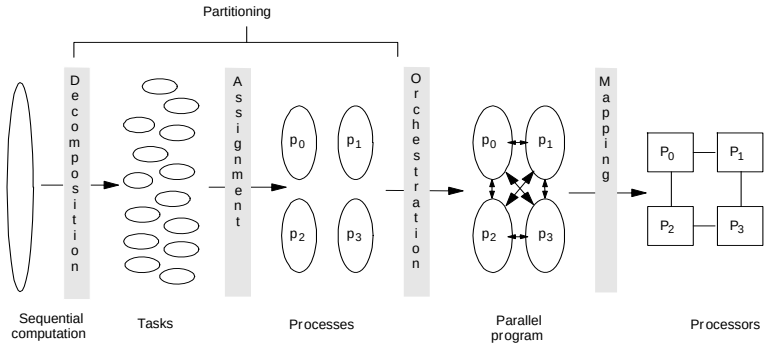
- Assumption: sequential algorithm is given
  - Sometimes we need a very different parallel algorithm
- Pieces of the job:
  - Identify work that can be done in parallel
    - Work includes computation, data access, and I/O
  - Partition work and perhaps data among processes
  - Manage data access, communication and synchronization
- Main goal: Speedup
  - Obtained with reasonable programming efforts and resource needs

$$Speedup(p) = \frac{Sequential\ execution\ time}{Parallel\ execution\ time\ (p)}$$

Definitions

- *Task*:
  - Arbitrary piece of work in a parallel program
  - Unit of concurrency that a parallel program can exploit
  - Executed sequentially; concurrency is only across tasks
  - Can be fine-grained or coarse-grained
  - Example: a single ray or a ray group in Raytrace
- *Process or thread*:
  - Abstract entity that performs the assigned tasks
  - Processes communicate and synchronize to perform their tasks
- *Processor*:
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - Write program in terms of processes, then map to processors

4 Steps in Creating a Parallel Program



- Decomposition of computation into tasks
- Assignment of tasks to processes
- Orchestration of data access, communication, and synchronization
- Mapping processes to processors

Decomposition

- Break up computation into a collection of tasks
  - Tasks may become available dynamically
  - Number of available tasks may vary with time
- Identify concurrency and decide level at which to exploit it
  - Fine-grained versus course-grained tasks
- Goal
  - Expose enough parallelism to keep processes busy
  - Too much parallelism increases the overhead of task management
- Number of parallel tasks available at a time is an upper bound on the achievable speedup

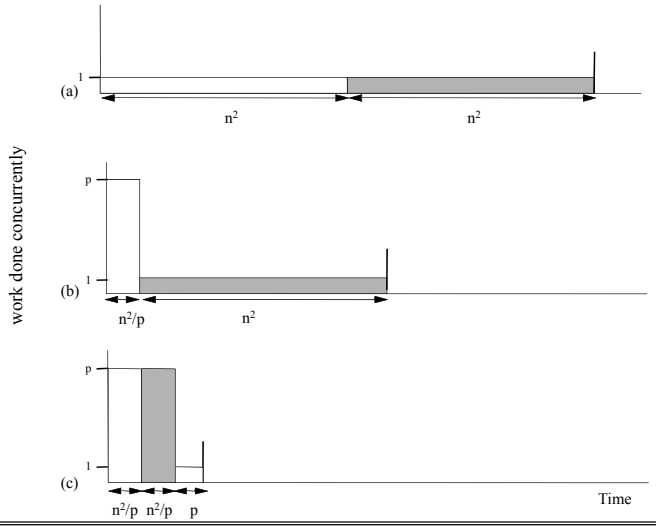
Limited Parallelism: Amdahl's Law

- Most fundamental limitation on parallel speedup
- If fraction *s* of execution time is serial then **speedup < *1/s***
  - *Reasoning: inherently parallel code can be executed in "no time" but inherently sequential code still needs fraction s of time*
- Example: if *s* is 0.2 then speedup cannot exceed 1/0.2 = 5.
- Using *p* processors $\text{Speedup} = \frac{1}{s + (1-s)/p}$
  - Total sequential execution time on a single processor is normalized to 1
  - Serial code on *p* processors requires fraction *s* of time
  - Parallel code on *p* processors requires fraction $(1 - s)/p$ of time

Example on Amdahl's Law

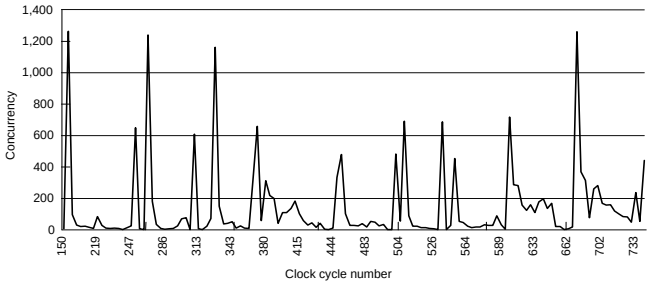
- Example: 2-phase calculation
  - Sweep over *n*-by-*n* grid and do some independent computation
  - Sweep again and add each value to global sum
- Time for first phase on *p* parallel processors = n^2/p
- Second phase serialized at global variable, so time = n^2
- Speedup <= $\frac{2n^2}{\frac{n^2}{p} + n^2} = \frac{2p}{p+1}$ or at most 2 for large *p*
- Improvement: divide second phase into two
  - Accumulate *p* private sums during first sweep
  - Add per-process private sums into global sum
- Parallel time is $n^2/p + n^2/p + p$, and speedup <= $\frac{2n^2p}{2n^2 + p^2}$

Understanding Amdahl's Law



Concurrency Profile

- Depicts number of available concurrent operations at each cycle
  - Can be quite irregular, *x*-axis is time and *y*-axis is amount of concurrency
- Is a function of the problem, input size, and decomposition
  - However, independent of number of processors (assuming unlimited)
  - Also independent of assignment and orchestration



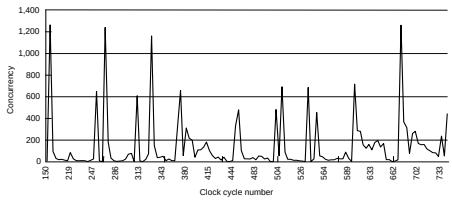
Generalized Amdahl's Law

- Area under concurrency profile is total work done
  - Equal to sequential execution time
- Horizontal extent is lower bound on time (infinite processors)

$$Speedup \leq \frac{Area\ under\ Concurrency\ Profile}{Horizontal\ Extent\ of\ Concurrency\ Profile}$$

- Let f_k be the number of x-axis points that have concurrency k

$$Speedup(p) \leq \frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil}$$



Assignment

- Assign or distribute tasks among processes
- Primary performance goals
  - Balance workload among processes – **load balancing**
  - Reduce amount of inter-process communication
  - Reduce the run-time overhead of managing the assignment
- Static versus dynamic assignment
  - Static is a predetermined assignment
  - Dynamic assignment is determined at runtime (reacts to load imbalance)
- Decomposition and assignment are sometimes combined
  - Called **partitioning** – As programmers we worry about partitioning first
  - Usually independent of architecture and programming model
  - Most programs lend themselves to structured approaches
  - But cost and complexity of using primitives may affect decisions

Orchestration

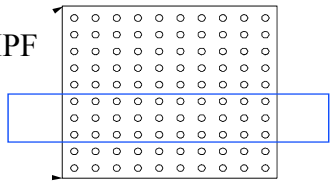
- Processes need mechanisms to …
  - Name and access data
  - Exchange data with other processes
  - Synchronize with other processes
- Architecture, programming model, and language play big role
- Performance Goals
  - Reduce cost of communication and synchronization
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Architects should provide appropriate and efficient primitives

Mapping

- Fairly specific to the system or programming environment
- Two aspects
  - Which process runs on which particular processor?
  - Will multiple processes run on same processor?
- Space-sharing
  - Machine divided into subsets, only one application runs at a time
  - Processes pinned to processors to preserve locality of communication
  - Time-sharing among multiple applications
- Dynamic system allocation
  - OS dynamically controls which process runs where and when
  - Processes may move around among processors as the scheduler dictates
- Real world falls somewhere in between

Partitioning Computation versus Data

- Typically we partition computation (decompose and assign)
- Partitioning data is often a natural choice too
  - Decomposing and assigning data to processes
- Decomposition of computation and data are strongly related
  - Difficult to distinguish in some applications like Ocean
  - Process assigned portion of an array is responsible for its computation
    - Know as **owner computes** arrangement
- Some programming languages like HPF
  - Allow programmers to specify the decomposition and assignment of data structures



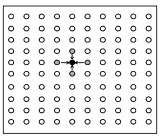
Goals of the Parallelization Process

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurr ency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninher ent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared r esources Schedule tasks to satisfy dependences early
Mapping	Yes	Put r elated pr ocesses on the same pr ocessor if necessary Exploit locality in network topology

Example: Iterative Equation Solver

- Simplified version of a piece of Ocean simulation
- Gauss-Seidel (near-neighbor)
  - Computation proceeds over a number of sweeps to convergence
  - Interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - New values of points above and left and old values below and right
  - Difference from previous value computed
  - Accumulate partial differences at end of every sweep
  - Check if has converged within a tolerance parameter
  - Standard sequential language augmented with primitives for parallelism
    - State of most real parallel programming today



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

Sequential Equation Solver Kernel

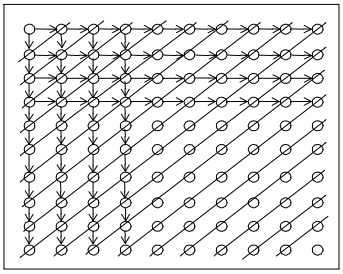
```
1.  int n;                                  /*size of matrix: (n + 2-by-n + 2) elements*/
2.  float **A, diff = 0;

3.  main()
4.  begin
5.     read(n) ;                            /*read input parameter: matrix size*/
6.     A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.     initialize(A);                       /*initialize the matrix A somehow*/
8.     Solve (A);                           /*call the routine to solve equation*/
9.  end main

10. procedure Solve (A)                     /*solve the equation system*/
11.    float **A;                           /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.    int i, j, done = 0;
14.    float diff = 0, temp;
15.    while (!done) do                     /*outermost loop over sweeps*/
16.       diff = 0;                         /*initialize maximum difference to 0*/
17.       for i ← 1 to n do                 /*sweep over nonborder points of grid*/
18.          for j ← 1 to n do
19.             temp = A[i,j];             /*save old value of element*/
20.             A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                A[i,j+1] + A[i+1,j]); /*compute average*/
22.             diff += abs(A[i,j] - temp);
23.          end for
24.       end for
25.       if (diff/(n*n) < TOL) then done = 1;
26.    end while
27. end procedure
```

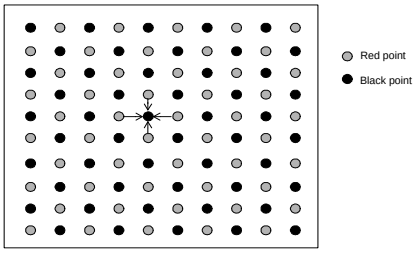
Decomposition

- Simple way to identify concurrency is to look at loop iterations
  - *Dependence analysis*; if not enough concurrency, then look further
- Not much concurrency here at loop level (all loops *sequential*)
  - Examine fundamental dependences
- Concurrency *O(n)* along anti-diagonals
- Serialization *O(n)* along diagonals
- Approach 1: Retain loop structure
  - Use point-to-point synchronization
  - Problem: Too much synchronization
- Approach 2: Restructure loops
  - Use global synchronization
  - Imbalance and too much synchronization



Exploit Application Knowledge

- Gauss-Seidel method is not an exact solution method
  - Results are not exact but within a tolerance value
- Sequential algorithm is not convenient to parallelize
  - Common sequential traversal (left to right and top to bottom)
  - Different parallel traversals are possible
- Reorder grid traversal: red-black
  - Red/black sweeps are each fully parallel
  - Different ordering of updates
  - May converge quicker or slower
  - Global synchronization between sweeps
    - Conservative but convenient
  - Longer kernel code



Fully Parallel Sweep Approach

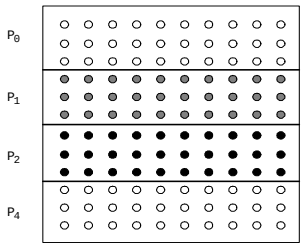
- Ignore dependences among grid points within a sweep
- Simpler than red-black ordering – **no ordering**
  - Convert nested sequential loops into parallel loops - **nondeterministic**
- Decomposition into elements: degree of concurrency is n^2

```
15.  while (!done) do              /*a sequential loop*/
16.    diff = 0;
17.    for_all i ← 1 to n do       /*a parallel loop nest*/
18.      for_all j ← 1 to n do
19.        temp = A[i,j];
20.        A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.          A[i,j+1] + A[i+1,j]);
22.        diff += abs(A[i,j] - temp);
23.      end for_all
24.    end for_all
25.    if (diff/(n*n) < TOL) then done = 1;
26.  end while
```

- Can also decompose into rows
  - Make inner loop (line 18) a sequential loop
  - Degree of concurrency reduced from n^2 to n

Assignment

- **`For_all`** decomposes iterations into parallel tasks
  - Leaves assignment to be specified later
- Static assignment: decomposition into rows
  - Block assignment of rows: Row i is assigned to process $\lfloor i / b \rfloor$
    - $b = block\ size = n / p$ (for n rows and p processes)
  - Cyclic assignment of rows: process i is assigned rows i, $i+p$, ...
- Concurrency is reduced from n to p
- Communication-to-Computation is reduced
  - Ratio is $O(p/n)$
- Dynamic assignment
  - Get a row index, work on row
  - Get a new row, ... etc.



Orchestration under Data Parallel Model

- Single thread of control operating on large data structures
- Needs primitives to partition computation and data
- G_MALLOC
  - Global malloc: allocates data in a **shared region** of the heap storage
- DECOMP
  - Assigns iterations to processes (task assignment)
    - [BLOCK, *, nprocs] means that first dimension (rows) is partitioned
      - Second dimension (cols) is not partitioned at all
    - [CYCLIC, *, nprocs] means cyclic partitioning of rows among nprocs
    - [BLOCK, BLOCK, nprocs] means 2D block partitioning among nprocs
    - [*, CYCLIC, nprocs] means cyclic partitioning of cols among nprocs
  - Also specifies how matrix data should be distributed
- REDUCE
  - Implements a reduction operation such as ADD, MAX, etc.
  - Typically implemented as a library function call

Data Parallel Equation Solver

```
1.   int n, nprocs ;                          /*grid size (n + 2-by-n + 2) and number of      processes*/
2.   float **A,  diff = 0;

3.   main()
4.   begin
5.     read(n); read( nprocs );       ;       /*read input grid size and number of      processes*/
6.     A ← G MALLOC  (a 2-d array of size n+2 by n+2 doubles);
7.     initialize(A);                         /*initialize the matrix A     somehow*/
8.     Solve (A);                             /*call the routine to solve     equation*/
9.   end main

10.  procedure Solve(A)                       /*solve the equation    system*/
11.      float **A;                           /*A is an (n + 2-by-n + 2)    array*/
12.    begin
13.    int i, j, done = 0;
14.    float mydiff = 0, temp;
14a.     DECOMP A[BLOCK,*,   nprocs];
15.    while (!done) do                       /*outermost loop over     sweeps*/
16.      mydiff = 0;                          /*initialize maximum difference to 0     */
17.      for_all  i ← 1 to n do               /*sweep over non-border points of      grid*/
18.        for all  j ← 1 to n do
19.          temp = A[ i,j];                  /*save old value of     element*/
20.          A[i,j]  ← 0.2 * (A[ i,j] + A[i,j-1] + A[i-1,j] +
21.            A[i,j+1] + A[i+1,j]);          /*compute  average*/
22.          mydiff = abs(A[ i,j] - temp);
23.        end  for_all
24.      end  for_all
24a.       REDUCE ( mydiff,  diff, ADD);
25.      if (diff/( n*n) < TOL) then done = 1;
26.    end while
27.  end procedure
```

Orchestration under Shared Memory

- Need primitives to create processes, synchronize them … etc.
- G_MALLOC(*size*)
  - Allocate shared memory of size bytes on the heap storage
- CREATE(*p*, *proc*, *args*)
  - Create *p* processes that start at procedure *proc* with arguments *args*
- LOCK(*name*), UNLOCK(*name*)
  - Acquire and release *name* for exclusive access to shared variables
- BARRIER(*name*, *number*)
  - Global synchronization among *number* of processes
- WAIT_FOR_END(*number*)
  - Wait for *number* of processes to terminate
- WAIT(*flag*), SIGNAL(*flag*) for event synchronization

Generating Threads

```
1.   int n, nprocs;          /*matrix dimension and number of processors to be used*/
2a.  float **A, diff;        /*A is global (shared) array representing the grid*/
                             /*diff is global (shared) maximum difference in current  sweep*/
2b.  LOCKDEC(diff_lock);     /*declaration of lock to enforce mutual exclusion*/
2c.  BARDEC (bar1);          /*barrier declaration for global synchronization between sweeps*/

3.  main()
4.  begin
5.     read(n); read(nprocs);   /*read input matrix size and number of processes*/
6.     A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.     initialize(A);           /*initialize A in an unspecified way*/
8a.    CREATE (nprocs-1, Solve, A);
8.     Solve(A);                /*main process becomes a worker too*/
8b.      WAIT_FOR_END (nprocs-1);  /*wait for all child processes created to terminate*/
9.  end main

10.  procedure Solve(A)
11.    float **A;               /*A is entire n+2-by-n+2 shared array,
                                  as in the sequential program*/
12. begin
13. ----
27. end procedure
```

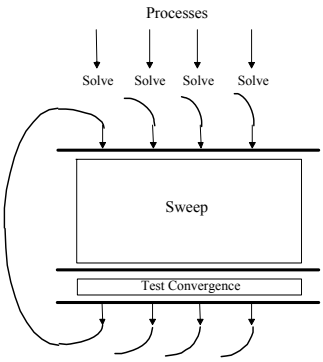
Equation Solver with Shared Memory

```
10.   procedure Solve(A)
11.       float **A;                          /*A is entire n+2-by-n+2 shared array,
                                                as in the sequential program*/
12.   begin
13.       int i,j, pid, done = 0;
14.       float temp, mydiff = 0;             /*private variables*/
14a.      int mymin = 1 + (pid * n/nprocs);   /*assume that n is exactly divisible by*/
14b.      int mymax = mymin + n/nprocs - 1    /*nprocs for simplicity here*/

15.       while (!done) do                    /*outer loop sweeps*/
16.         mydiff = diff = 0;                /*set global diff to 0 (okay for all to do it)*/
16a.        BARRIER(bar1, nprocs);            /*ensure all reach here before anyone modifies diff*/
17.         for i ← mymin to mymax do         /*for each of my rows*/
18.           for j ← 1 to n do               /*for all nonborder elements in that row*/
19.               temp = A[i,j];
20.               A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                                 A[i,j+1] + A[i+1,j]);
22.               mydiff += abs(A[i,j] - temp);
23.           endfor
24.         endfor
25a.        LOCK(diff_lock);                          /*update global diff if necessary*/
25b.          diff += mydiff;
25c.        UNLOCK(diff_lock);
25d.        BARRIER(bar1, nprocs);            /*ensure all reach here before checking if done*/
25e.        if (diff/(n*n) < TOL) then done = 1;  /*check convergence; all get
                                                    same answer*/
25f.        BARRIER(bar1, nprocs);
26.       endwhile
27.   end procedure
```


Notes on the Parallel Program

- Single Program Multiple Data (SPMD)
  - Not lockstep or even necessarily exact same instructions as in SIMD
  - May follow different control paths through the code
- Unique *pid* per process
  - Numbered as 0, 1, …, *p*-1
  - Assignment of iterations to processes is controlled by loop bounds
- Barrier Synchronization
  - To separate distinct phases of computation
- Mutual Exclusion
  - To accumulate sums into shared *diff*
- Done condition
  - Evaluated redundantly by all



Mutual Exclusion

- Race Conditions

P1	P2	
r1 ← diff		{P1 gets diff in its r1}
	r1 ← diff	{P2 also gets diff in its r1}
r1 ← r1+mydiff		
	r1 ← r1+mydiff	
diff ← r1		{race conditions}
	diff ← r1	{sum is Not accumulated}

- LOCK-UNLOCK around *critical section*
  - Ensure exclusive access to shared data
  - Implementation of LOCK/UNLOCK must be atomic
- Locks are expensive
  - Cause contention and serialization
  - Associating different names with locks reduces contention/serialization

Event Synchronization

- Global Event Sychronization
  - BARRIER(*name*, *nprocs*)
    - Wait on barrier *name* until a total of *nprocs* processes have reached this barrier
    - Built using lower level primitives
    - Often used to separate phases of computation
    - Conservative form of preserving dependences, but easy to use
  - WAIT_FOR_END (*nprocs*-1)
    - Wait for *nprocs*-1 processes to terminate execution
- Point-to-Point Event Synchronization
  - One process notifies another of an event so it can proceed
    - Common example: produce-consumer
  - WAIT(*sem*) and SIGNAL(*sem*) on a semaphore *sem* with **blocking**
  - Use of ordinary shared variables as flags with **busy-waiting** or **spinning**

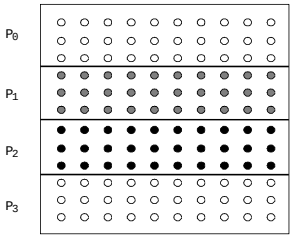
Orchestration Under Message-Passing

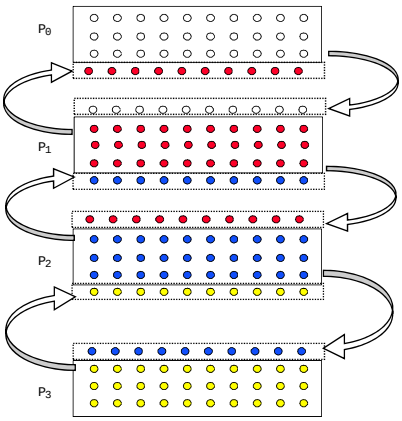
- Cannot declare *A* to be a global shared array
  - Compose it logically from per-process private arrays
  - Usually allocated in accordance with the assignment of work
    - Process assigned a set of rows allocates them locally
- Transfers of entire rows between traversals
- Structured similar to SPMD in Shared Address Space
- Orchestration different
  - Data structures and data access/naming
  - Communication
  - Synchronization
- Ghost rows

Basic Message-Passing Primitives

- CREATE(*procedure*)
  - Create process that starts at *procedure*
- SEND(*addr*, *size*, *dest*, *tag*), RECEIVE(*addr*, *size*, *src*, *tag*)
  - Send *size* bytes starting at *addr* to *dest* process with *tag* identifier
  - Receive a message with *tag* identifier from *src* process and put *size* bytes of it into buffer starting at *addr*
- SEND_PROBE(*tag*, *dest*), RECEIVE_PROBE(*tag*, *src*)
  - *Send_probe* checks if message with *tag* has been sent to process *dest*
  - *Receive_probe* checks if message with *tag* has been received from *src*
- BARRIER(*name*, *number*)
- WAIT_FOR_END(*number*)

Data Layout and Orchestration





Data partition allocated per processor

Add ghost rows to hold boundary data

Send edges to neighbors

Receive into ghost rows

Compute as in sequential program

Equation Solver with Message Passing

```
10.  procedure Solve()
11.  begin
13.    int i,j, pid, n' = n/nprocs, done = 0;
14.    float temp, tempdiff, mydiff = 0;        /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
7.   initialize(myA)                           /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.    mydiff = 0;                             /*set local diff to 0*/
     /* Exchange border rows of neighbors into myA[0,*] and myA[n'+1,*]*/
16a.   if (pid != 0) then SEND(&myA[1,1],n*sizeof(float),pid-1,ROW);
16b.   if (pid != nprocs-1) then
         SEND(&myA[n',1],n*sizeof(float),pid+1,ROW);
16c.   if (pid != 0) then RECEIVE(&myA[0,1],n*sizeof(float),pid-1,ROW);
16d.   if (pid != nprocs-1) then
         RECEIVE(&myA[n'+1,1],n*sizeof(float), pid+1,ROW);
17.    for i ← 1 to n' do                      /*for each of my (nonghost) rows*/
18.      for j ← 1 to n do                     /*for all nonborder elements in that row*/
19.        temp = myA[i,j];
20.        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.          myA[i,j+1] + myA[i+1,j]);
22.        mydiff += abs(myA[i,j] - temp);
23.      endfor
24.    endfor
```

Equation Solver with Message Passing

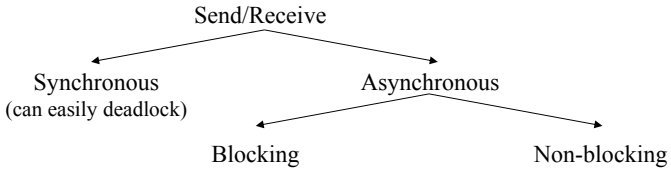
```
            /*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
25a.        if (pid != 0) then                          /*process 0 holds global total diff*/
25b.          SEND(mydiff,sizeof(float),0,DIFF);
25c.          RECEIVE(done,sizeof(int),0,DONE);
25d.        else                                        /*pid 0 does this*/
25e.          for i ← 1 to nprocs-1 do                  /*for each other process*/
25f.            RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.            mydiff += tempdiff;                     /*accumulate into total*/
25h.          endfor
25i           if (mydiff/(n*n) < TOL) then  done = 1;
25j.          for i ← 1 to nprocs-1 do                  /*for each other process*/
25k.            SEND(done,sizeof(int),i,DONE);
25l.          endfor
25m.      endif
26.     endwhile
27.     end procedure
```


Notes on Message Passing Program

- Use of ghost rows
- Receive does not transfer data, send does
  - Unlike Shared Memory which is usually receiver-initiated (load fetches data)
- Communication done at beginning of iteration - deterministic
- Communication in whole rows, not element at a time
- Core similar, but indices/bounds in local rather than global address space
- Synchronization through sends and receives
  - Update of global diff and event synchronization for done condition
  - Mutual exclusion: only one process touches data
  - Can implement locks and barriers with messages
- Can use REDUCE and BROADCAST library calls to simplify code

```
            /*communicate local diff values and determine if done, using reduction and broadcast*/
25b.        REDUCE (0,mydiff,sizeof(float),ADD);
25c.        if (pid == 0) then
25i.          if (mydiff/(n*n) < TOL) then done = 1;
25k.          endif
25m.        BROADCAST (0,done,sizeof(int),DONE);
```

Send and Receive Alternatives



- Synchronous Send returns control to calling process only when
  - Corresponding synchronous receive has completed successfully
- Synchronous Receive returns control when
  - Data has been received into destination process's address space
  - Acknowledgement to sender
- Blocking Asynchronous Send returns control when
  - Message has been taken from sending process buffer and is in care of system
  - Sending process can resume much sooner than a synchronous send
- Blocking Asynchronous Receive returns control when
  - Similar to synchronous receive, but No acknowledgment to sender

Send and Receive Alternatives – cont'd

- Non-blocking Asynchronous Send returns control immediately
- Non-blocking Receive returns control after posting intent to receive
  - Actual receipt of message is performed asynchronously
- Separate primitives to probe the state of non-blocking send/receive
- Semantic flavors of send and receive affect …
  - When data structures or buffers can be reused at either end
  - Event synchronization (non-blocking send/receive have to use probe before data)
  - Ease of programming and performance
- Sending/receiving non-contiguous regions of memory
  - Stride, Index arrays (gather on sender and scatter on receiver)
- Flexibility in matching messages
  - Wildcards for tags and processes

Orchestration: Summary

- Shared address space
  - Shared and private data explicitly separate
  - Communication implicit in access patterns
  - No *correctness* need for data distribution
  - Synchronization via atomic operations on shared data
  - Synchronization explicit and distinct from data communication
- Message passing
  - Data distribution among local address spaces needed
  - No explicit shared structures (implicit in communication patterns)
  - Communication is explicit
  - Synchronization implicit in communication
  - Mutual exclusion when one process modifies global data

Grid Solver Program: Summary

	Shared Address	Message Passing
Explicit global data structure?	**Yes**	**No**
Assignment independent of data layout?	**Yes**	**No**
Communication	**Implicit**	**Explicit**
Synchronization	**Explicit**	**Implicit**
Explicit replication of border rows?	**No**	**Yes**

- Decomposition and Assignment
  - Similar in shared memory and message-passing
- Orchestration is different
  - Data structures, data access/naming, communication, synchronization